Finite Set Theory in ACL2

J Strother Moore*

May 25, 2000

Abstract

We describe the ACL2 book `books/finite-set-theory/set-theory`, which provides the primitives of finite set theory in ACL2. The book supports finite sets of ACL2 objects, including sets of sets. Among the primitives defined in the `"S"` package are set equality, denoted `=`, set membership, `mem`, the subset relation, `subsetp`, set manipulation functions such as `union`, `intersection` and `diff`, a choice function, `choose`, and others. The book provides lemmas about these primitives, although as of this writing the book is not as complete as I would like. In addition, the book provides various strategies, implemented as macros, for proving theorems in set theory. An example of a strategy is to prove that two sets are equal by proving that `e` is an element of one precisely when it is an element of the other. The report has four appendices dealing with recursion on sets, the background of the project, discussion of some dead ends explored, and a list of some of the nice theorems proved here.

Contents

*Department of Computer Sciences, University of Texas, Austin, TX 78712, moore@cs.utexas.edu. This research was supported in part by Compaq Systems Research Center, Palo Alto, CA.

1 Overview

Doesn't ACL2 already provide sets? It contains such standard Lisp functions as `member`, `subsetp`, `union` and `intersection`. Granted, these operate on lists. But they ignore duplication and order and hence essentially provide finite sets. Right?

Wrong. Taking the "sets as lists" view, `'(1 2)` is the representation of $\{1, 2\}$. `'(2 1)` also represents that set. Each of these is a `subsetp` of the other and so there is a sense in which we "know" they represent the same set. Indeed, we can write $\{1, 2\}$ as $\{2, 1\}$. They are just different notations for the same object. But `'(1 2)` and `'(2 1)` are different Lisp objects.

Now, is the set $\{1, 2\}$ a member of the set $\{\{1, 2\}\}$? The answer is obviously yes. But `(member '(1 2) '((2 1)))` is "false." That is because Lisp sense of equality is "too" strong and that sense of equality pervades the definition of both membership and subset.[1]

Having established the need to do *something* to provide first-class finite sets in ACL2, here is what we will do.

We are going to develop what might be called the "hereditarily finite sets built from the ACL2 universe of objects." That is, the base elements, or "ur-elements," of our sets will be the ACL2 objects and the elements of all sets will be these ur-elements or other such sets.

But we cannot extend the universe of ACL2 objects with a new kind of object called "set". Instead, non-empty sets are represented by lists and we define "set theory" functions that treat these lists as though they were sets. To allow lists to be ur-elements it is necessary to provide an "escape convention" whereby the set interpretation of a list is abandoned. These and other representational issues are largely irrelevant to the end user.

A given set may be represented by several distinct lists. ACL2's basic equality relation, `equal`, can thus distinguish two lists that represent the same set.[2] Other ACL2 functions, like `car` and `cdr`, may also distinguish such lists.

As part of our set theory work, a new equivalence relation, `=`, is defined. This relation is the equality relation in the universe consisting of the hereditarily finite sets built from the ACL2 universe of objects. We can distinguish the ur-elements with `=`; that is, `=` is not just the equality relation between sets. So forget about ACL2's native `equal`. We adopt `=` as the basic sense of equality here.

Ur-elements are distinct from sets. For example, `2` not a set.

The primitive set theory predicates and functions, such as member and union, are defined so that `=` is a congruence relation for each argument position. A unary function `f` admits `=` as a congruence relation if a `=` b implies that `f`(a) `=` `f`(b), or, in the ACL2 syntax `(implies (= a b) (= (f a) (f b)))`.

Such functions cannot distinguish the multiple representatives of a set. As long as formulas are constructed entirely from such function symbols, the fact that there are multiple representatives is undetectable. We can thus speak of s "being" a set or of sets x and y "being equal" when what we actually mean is that s represents a set and that x and y represent the same set. In the main body of this report we confine our attention to such set theory functions and formulas.

Whenever a new "set theory" function symbol is defined it is crucial to prove that it admits `=` as a congruence relation for each argument. This is generally not difficult provided the new symbol is defined in terms of other such "set theory" functions; in particular, new symbols should not be defined in terms of native ACL2 functions like `equal` and `car` (unless you want to expose the representation of sets as lists and argue from first principles that your new definition nevertheless admits `=` as a congruence).

[1] Note the mutual recursion inherent in these three concepts. membership is defined in terms of equality, subset is defined in terms of membership, and equality is defined in terms of subset. I explored a direct approach along these lines. See Appendix C.

[2] In ACL2, "relations" and "predicates" are defined as Boolean valued functions, i.e., functions that return either `t` or `nil`. Our uses of the words "relation" and "predicate" here are synonymous with "Boolean valued function." When we make general remarks about functions those remarks apply to relations and predicates too.

2 ACL2 Set Theory

Definition. An *object* is either an ur-element or a finite set of objects.

Definition. An *ur-element* is

- a number (i.e., a rational or complex rational),
- one of 255 distinct character objects,
- a string of character objects,
- a symbol[3] (composed of a package name and a symbol name, each of which is a string), or
- a cons of the form `(:UR-CONS (`x `.` y`))`.

Observe that if a Lisp cons pair, `(`x `.` y`)`, is to be used as an ur-element, it must be so marked by embedding it in an `:ur-cons` form. Here, x and y are arbitrary Lisp objects and are not treated as sets.

The empty set is the ur-element `nil`. Note that according to our taxonomy, one ur-element is also a set, namely `nil`.

Definition. A *set* is an object that is either `nil` or not an ur-element.

In this report, I specify key functions in the ACL2 set theory "book". These functions are all defined within a new symbol package, named `"S"` for "sets." Here are the first two functions documented.

`(ur-elementp` a`)`: `t` or `nil` according to whether a is an ur-element.

`(setp` a`)`: `t` or `nil` according to whether a is a set.

Because ACL2's is a syntactically untyped language it is possible to use ur-elements where sets are expected. We deal with this with a sweeping convention.

The Non-Set Convention. If a non-`nil` ur-element is used where a set is expected, all set theory functions shall behave as though the empty set had been used instead.

For example, our answer to the question "does `3` occur as an element in the set `5`?" is "no," because `5` is not a set and hence the question is treated as though it had been "does `3` occur as an element in the set {}?" Occasionally below we note the impact of the non-set convention. But in general this should not concern the reader since such "ill-formed" questions are never asked.

3 Notation for Literal Constants

Ur-elements other than conses are written as in Lisp. Here are examples of the first four kinds of ur-elements:

- numbers: `1`, `-123`, `22/7`, `#c(1/2 1/3)`;
- characters: `#\A`, `#\a`, `#\Space`
- strings: `"Write"`, `"Hello World`"

[3]Other than the symbol `:UR-CONS` in the keyword package.

- symbols: `X`, `MYPKG::MAP`, `PRIME-LIST`

When a cons pair is used as an ur-element it must be within an `:ur-cons` form. Here are three examples.

- The list containing the symbolic names of the first three days of the week: `(:UR-CONS (SUNDAY MONDAY TUESDAY))`.
- The digits, as integers, `(:UR-CONS (0 1 2 3 4 5 6 7 8 9))`.
- An ACL2 association list pairing the symbol `A` to 22 and the symbol `B` to -12, `(:UR-CONS ((A . 22) (B . -12)))`.

Generally, we expect that the set theory user will use sets for collections. However, the ur-elements collected might be objects represented by lists, such as machine states, etc. These cons pairs may be used as ur-elements but must be marked.

Non-empty sets are written as lists of their elements. For example:

- The set containing the symbolic names of the first three days of the week, which ordinarily might be written {`SUNDAY`, `MONDAY`, `TUESDAY`}, may be written `(SUNDAY MONDAY TUESDAY)`.
- The set of digits, as integers, may be written `(0 1 2 3 4 5 6 7 8 9)`.

Duplications and order are irrelevant to the set denoted.

Convention. If a non-empty list is used as a set, it denotes the set obtained by adding the object denoted by its car to the set denoted by its cdr.

Observation. When used as sets, all Lisp atoms denote the empty set and all Lisp conses denote non-empty sets.

Following the normal rules of Lisp, it is necessary to quote certain literal constants when they are used in terms. In particular, symbols, and parenthesized notation – i.e., ur-conses and non-empty sets – must be quoted to be used as literal constants.

The set $\{1, 2, 3\}$ can be written `(1 2 3)` in ACL2 set theory. But to use that as a literal constant in a set theory expression one must quote it. For example, to denote the result of applying the function `max` to the above set, one would write `(max '(1 2 3))`, not `(max (1 2 3))`.[4]

The convention on the interpretation of a cons pair as a set can be phrased more clearly as:

Convention. When the literal constant `'(`α `.` β`)` is used as a set, the set denoted is $\{'\alpha\} \cup {'\beta}$. The non-set convention applies to β.

Thus `'(1 2 . 3)`, which might also be written in Lisp as `'(1 . (2 . 3))` denotes the set $\{1, 2\}$, since the non-set ur-element `3` is being used where a set is expected.

4 Primitive Set Building Functions

`(scons` e a`)`: $\{e\} \cup a$, i.e., the set obtained by adding e as an element to the set a. The second argument is expected to be a set.

For example, `(scons (+ x y) (scons (* x y) nil))` is the set which might be written $\{x + y, xy\}$. If `a` is the set $\{1, 2, 3\}$, then `(scons 4 a)` is $\{1, 2, 3, 4\}$ and `(scons 2 a)` is the same as a.

[4] The function symbol `max` is not defined in this work.

	term	*value*
1.	`(= 1 1)`	`T`
2.	`(= 1 "Hi")`	`NIL`
3.	`(= '(1 2 3) '(3 1 2 1))`	`T`
4.	`(= '(1 2 3) '(3 1 1))`	`NIL`
5.	`(= '((1 2)) '((2 1)))`	`T`
6.	`(= 1 '(2 3))`	`NIL`
7.	`(mem 1 '(1 2 3))`	`T`
8.	`(mem "Hi" '(1 2 3))`	`NIL`
9.	`(mem '(2 1) '(1 (1 2) "Hi"))`	`T`
10.	`(mem 1 3)`	`NIL`
11.	`(mem 3 '(1 2 . 3))`	`NIL`
12.	`(mem 2 '(1 2 . 3))`	`T`
13.	`(subsetp '(1 2) '(2 1))`	`T`
14.	`(subsetp '(1 2 3) '(2 1))`	`NIL`
15.	`(subsetp '(1 2) '(-7 2 0 1 3))`	`T`
16.	`(subsetp 3 '(2 1))`	`T`
17.	`(subsetp '(2 1) 3)`	`NIL`

Table 1: Some Examples of Equality, Membership, and Subset

5 Roster Notation

Consider $\{x+y, xy\}$. This denotes the set whose elements are the values of the two expressions given. This is called *roster notation* and is different from the notation for literal constants because in roster notation the elements are not necessarily constants. In ACL2 set theory we write `(brace (+ x y) (* x y))` to denote $\{x+y, xy\}$.

`(brace` α_1 ... α_k`)`: the set whose elements are given by the values of the k expressions. `(Brace` α_1 ... α_k`)` is just a macro that expands into `(scons` α_1 `(scons` α_2 ... `(scons` α_k `nil)...))`

`(brace*` α_1 ... α_k β`)`: the set obtaining by unioning `(brace` α_1 ... α_k`)` and β. `(Brace*` α_1 ... α_k β`)` is just a macro that expands into `(scons` α_1 `(scons` α_2 ... `(scons` α_k β`)...))`

The set `(Brace` $\alpha_1 ... \alpha_k$`)` may not contain as many as k elements. For example, if `x` is `0`, then `(brace (+ x y) (* x y))` is the singleton set containing `0`, as is the literal constant `'(0 0)`.

6 Equality, Membership and Subset

`(=` a b`)`: If a and b are the same object, then `t`, otherwise, `nil`.

`(mem` e a`)`: $e \in a$. The second argument is expected to be a set. The result is `t` or `nil`.

`(subsetp` a b`)`: $a \subseteq b$. Both arguments are expected to be sets. The result is `t` or `nil`.

Examples 1 and 2 of Table 1 illustrate that `=` distinguishes ur-elements from one another. Examples 3 and 4 illustrate that `=` distinguishes sets from one another, and example 3 additionally illustrates that the order and number of occurrences of the elements in presentation of a set are irrelevant. Example 5 illustrates that elements of sets are compared with `=`. Example 6 illustrates that `=` distinguishes ur-elements from sets.

These examples also illustrate that the relation denoted by `=` is not just "set equality" or "ur-element equality" but "object equality." That is, there is one sense of equality here.

Examples 7–9 illustrate the expected properties of `mem`. Note that example 9 illustrates that `mem` compares elements using `=`, since it correctly identifies the set `(2 1)` as the element `(1 2)`. Examples 10–12 illustrate the non-set convention as it applies to `mem`.

Examples 13–15 illustrate the expected properties of `subsetp`. Examples 16 and 17 illustrate the non-set convention as it applies to `subsetp`.

7 Arithmetic

The arithmetic functions of ACL2 set theory are exactly the arithmetic functions of ACL2. That is, the symbols for the arithmetic functions are imported into the `"S"` package. Some of the common ones are listed below for convenience. Except for `=`, which has already been discussed but is included below for completeness, when non-numeric arguments are provided, the functions act as though `0` had been provided.

`(-` i`)`: the negation of i.

`(+` i j`)`: the sum of i and j.

`(-` i j`)`: the difference of i and j.

`(*` i j`)`: the product of i and j.

`(/` i j`)`: the quotient of i and j.

`(=` i j`)`: `t` or `nil` according to whether i and j are the same.

`(<` i j`)`: `t` or `nil` according to whether $i < j$.

`(<=` i j`)`: `t` or `nil` according to whether $i \leq j$.

In addition, ACL2 set theory includes the following functions that produce numbers.

`(cardinality` a`)`: $|a|$. The argument is expected to be a set. The value is an ur-element (a natural number).

`(nats` n`)`: the set of natural numbers from `0` through n, inclusive. If n is not a natural number, the set contains just `0`.

8 Basic Operations on Sets

`(union` a b`)`: $a \cup b$. Both arguments are expected to be sets.

`(intersection` a b`)`: $a \cap b$. Both arguments are expected to be sets.

`(diff` a b`)`: $a \setminus b$. Both arguments are expected to be sets.

Other operations could easily be defined.

9 A Choice Function

In order to allow us to define certain functions on sets, such as extracting the components of an ordered pair represented by $\{x, \{x, y\}\}$, we must be able to obtain an element of a non-empty set. We therefore define the function:

`(choose` a`)`: an element of the set a, if a is non-empty. The argument is expected to be a set.

Key properties of `(choose s)` are that it admits = as a congruence, i.e., the choice is independent of the representation or presentation of `s`, and `(choose s)` is a member of `s` if `s` is non-empty. Ideally, perhaps, nothing else need be said about `choose`. But it is in fact quite specifically defined. We simply opt seldom to expose its other properties. But logically speaking those properties are present and important.

`Choose` is computable. For example, `(choose '(1 2 3 4))` is `4`.

In fact, `choose` chooses the largest element of the set, where the ordering is a certain (arbitrarily defined but unspecified in this report) total ordering on our objects.

Our definition of `choose` allows us to prove the following property.

```
Weak Choose-Scons Property:
(or (= (choose (scons e a)) e)
    (= (choose (scons e a)) (choose a)))
```

That is, `choose` on `(scons e a)` either returns `e` or its choice from `a`. Of course, it is possible to say exactly which of these cases obtains: `(choose (scons e a))` is `e`, if `e` dominates `(choose a)`, and is `(choose a)` otherwise. We call this stronger statement the *Strong Choose-Scons Property.* The stronger property introduces the notion of the total ordering while the weak property does not. Therefore, we prefer to use only the weak property.

Suppose it is desired to have an undefined choice function that is known only to admit = as a congruence and to select a member of a non-empty set. The following will introduce `ch` as such a function.

```
(encapsulate ((ch (s) t))
 (local (defun ch (s) (choose s)))
 (defcong = = (ch s) 1)
 (defthm mem-ch
   (implies (not (ur-elementp s))
            (mem (ch s) s))))
```

It should be noted that such an undefined `ch` does not necessarily enjoy the analogue of the weak `choose-scons` property. That is, it is impossible to prove from the properties of `ch` above that `(ch (scons e s))` is either `e` or `(ch s)`. For example, suppose `ch` chooses the largest element of the set if the cardinality of the set is odd and chooses the smallest element if the cardinality is even. Then `ch` would have the required properties and furthermore `(ch (scons 1 '(2 3)))` would be `3`, which is neither `1` nor the `(ch '(2 3))`, which is `2`.

Therefore, even when operating only with the weak `choose-scons` property, we are assuming more about `choose` than we could an arbitrary choice function. Most importantly, our `choose` is executable, which means that functions defined in terms of it are also executable. Such functions include `pair`, `hd`, `tl` and `apply`, discussed below.

10 Ordered Pairs as Sets

`(pair` x y`)`: a set representing the ordered pair $\langle x, y\rangle$. We call such a set simply a *pair*. `(Pair` x y`)` is defined to be `(brace` x `(brace` x y`))`.

(pairp a): t or nil according to whether a is a pair.

(hd a): if a is the pair $\langle x, y\rangle$, then x; otherwise, nil.

(tl a): if a is the pair $\langle x, y\rangle$, then y; otherwise, nil.

(pairps s): t or nil according to whether s is a set of pairs. The argument is expected to be a set.

At the moment, we only use ordered pairs to develop the notion of functions. It is possible to arrange things so that set theoretic ordered pairs are as convenient for the set theory user as ACL2's native cons pairs are for the ACL2 user.

In a separate book, set-count.lisp, I define a function that measures the size of a set, taking into account the size of the elements. I prove that the hd and tl of a pair have smaller set-counts than the pair. This means it is possible to define recursive functions that recur by hd and tl, using set-count as the justifying measure. After defining a function that recurs this way, it is, of course, necessary to prove that it admits = as a congruence. That can be done conveniently using a method implemented in set-count.lisp. Once these preliminaries are done, it is convenient to prove theorems about recursive pair processing functions. For example, below I define a version of Lisp's append that uses ordered pairs. This function, of course, generally constructs a set.

```
(defun pair-append (x y)
  (declare (xargs :measure (set-count x)))
  (if (pairp x)
      (pair (hd x)
            (pair-append (tl x) y))
    y)).
```

I prove that it admits = as a congruence, using the following two commands.

```
(defcong = = (pair-append x y) 1
  :hints (("Goal" :induct (pair-equal-hint x acl2::x-equiv))))

(defx :strategy :congruence (pair-append x y) 2)
```

We could dress up the former command as a macro.

With these preliminaries out of the way, the system easily proves the associativity of pair-append with a proof exactly analogous to that for Lisp's append.

```
(defthm pair-append-is-associative
  (= (pair-append (pair-append a b) c)
     (pair-append a (pair-append b c)))).
```

11 Functions

In our set theory, functions are sets of ordered pairs.

(functionp f): if f is a set of pairs and no two elements of f have the same hd, then t; otherwise, nil. If a function f contains (pair e v), then we say v is the *value* of f on e.

(domain f): $\{$(hd x) $\mid x \in f\}$, or, to be more formal, $\{e \mid \exists x(x \in f \wedge e =$(hd x) $)\}$. The argument is expected to be a set.

(range f): $\{$(tl x) $\mid x \in f\}$. The argument is expected to be a set.

`(apply` f e`)`: if f is a function and e is in its domain, then the value of f on e. More generally, let s_e be $\{p \mid p \in s \wedge (\texttt{hd}\ p) = e\}$. Then `(apply` f e`)` is defined to be `(tl (choose` s_e`))`. The first argument is expected to be a set. Note that if a function is applied to an argument outside its domain, the result is `nil`. The behavior of `apply` on non-functions is determined by `choose`.

`(except` f e v`)`: the set obtained by removing from f all elements with `hd` e and then adding `(pair` e v`)`. If f is a function then `(except` f e v`)` is the function that is everywhere equal to f except on e where the value is v.

`(restrict` f a`)`: The image of f on those elements of a in the domain of f. Both arguments are expected to be sets. `Restrict` always produces a function.

`(func (`e_1 v_1`) ... (`e_n v_n`))`: the function mapping e_i to v_i. Presentation order is used to resolve conflicts. That is, if e_i is e_j for some $i < j$, then the function maps e_i to v_i.

`(func* (`e_1 v_1`) ... (`e_n v_n`)` f`)`: f except at e_i where the result is v_i. Presentation order is used to resolve conflicts.

12 Sequences

`(sequencep` s`)`: `t` if s is a function whose domain is `(diff (nats (cardinality` s`)) '(0))`; otherwise, `nil`.

`(shift` i j f d`)`: If i and j are integers, then the function obtained by mapping $k + d$ to `(apply` f k`)`, for every $i \leq k \leq j$. If i or j is not an integer, the result is `nil`.

`(concat` r s`)`: `(union` r `(shift 1 (cardinality` s`)` s `(cardinality` r`)))`.

13 Sample Theorems

In Appendix D some of the theorems available in the set theory book are listed to illustrate proved properties of the foregoing definitions.

These theorems have some subtle attractive properties that one can appreciate only by considering alternative formulations of set theory in ACL2. One is that they are not burdened by hypotheses restricting them to sets or objects. Another is the use of `=` as the only sense of equivalence.[5] Any theorem concluding with an `=` could be used as a rewrite rule (under certain restrictions imposed by ACL2's rewriter) to normalize set theory expressions. A third is that the proofs of most of these theorems are extraordinarily simple, encouraging me to think that the work can be elaborated.

14 Codified Proof Strategies

The set theory book includes several proof strategies particular to set theory. Such strategies are convenient because they overcome ACL2's lack of quantification. We discuss two examples and then explain this remark.

Suppose you wish to prove `(=` α β`)`, where α and β are two set theory expressions that produce sets (as opposed to ur-elements). Sometimes the easiest way to prove such a theorem is to prove that $(e \in \alpha) \leftrightarrow$

[5] Virtually all of the theorems in the set theory work use `=`. There are a few instances where `equal` or `iff` had to be used even though `=` is logically sufficient. Future work will have to identify the cause of this problem — probably a missing congruence rule for some basic function — so that the set theory user need not be aware of ACL2's native equality.

($e \in \beta$). That is, a set is entirely determined by its elements. This strategy can be generalized so that one can prove `(implies` γ `(=` α β`))` analogously. We implement the command

```
(defx name
  (implies γ (= α β))
  :strategy :set-equivalence)
```

so that the indicated formula is proved (and given the indicated *name*) by proving two lemmas, namely, `(implies (and` γ `(mem e` α`)) (mem e` β`))` and the symmetric formula.

Another special strategy is useful when α and β are functions: prove that applying them produces identical results. We provide

```
(defx name
  (= α β)
  :strategy :functional-equivalence)
```

to be used when α and β yeild functions. The strategy proves the formula by proving (a) `(functionp` α`)`, (b) `(functionp` β`)`, (c) `(= (domain` α`) (domain` β`))`, and (d)

```
(implies (mem e (domain α))
         (= (apply α e)
            (apply β e))).
```

Proof obligation (d) could be simplified by dropping the hypothesis; we included the hypothesis simply because it weakens the proof obligation.

The functional equivalence strategy is generalized to handle implications, `(implies` γ `(=` α β`))`, as illustrated with the `:set-equivalence` strategy.

These strategies would not be necessary in a logic providing full first-order quantification.[6] For example, condition (d) could be expressed by universally quantifying over `e` and then (a)–(d) could be conjoined and shown to imply `(=` α β`)`. The lack of quantification in ACL2 prevents writing the quantified form of (d) as a hypothesis.

`Defx` is implemented with `encapsulate` and functional instantiation. That is, we constrain two functions, `alpha` and `beta` to satisfy (a)–(d) and then prove that they are identical. The `:functional-equivalence` strategy then does a functional instantiation of this result, generating (a)–(d) as proof obligations. The theorem that `concat` is associative is actually proved using this strategy. The proof command is

```
(defx associativity-of-concat
  (implies (and (sequencep a)
                (sequencep b)
                (sequencep c))
           (= (concat (concat a b) c)
              (concat a (concat b c))))
  :strategy :functional-equivalence)
```

That is, the two `concat` expressions are proved to be functions with the same domains and to be equal when applied.

Other strategies can be implemented so the end user can give proof hints to ACL2 in set theoretic terms.

[6] Actually, ACL2 does provide full first-order quantification via `defun-sk`. We could implement this strategy that way. But it would still require the user's explicit direction to use it.

15 Recursive Functions on Sets

The first order nature of ACL2, combined with the absence of quantification, prevents the formalization of set comprehension in its general form. That is, it is impossible in ACL2 formalize with complete generality such notation as "$\{x \mid \phi(x)\}$." We have implemented some macros to mitigate the problem somewhat.

The first step is to consider only notation such as $\{x \mid x \in s \rightarrow \phi(x)\}$, where s is a (finite) set. It is then possible to map over s with a recursive function to identify the appropriate elements. But because ACL2 is first-order, we cannot define a function that takes ϕ as an argument. So the second step is to define a recursive function, f, for any given ϕ, to compute the above set from s.

The most obvious disadvantages of this approach are (a) one must introduce a function symbol f to capture each use of set-builder notation, (b) one must prove "the same" theorems about each such f, and (c) one must prove theorems that relate any two such f's that become entwined in the same problem. Much of this can be handled by macros and in the translator from set theory to ACL2. We therefore deal with the most fundamental issues here.

The general scheme for defining a recursive function to extract the ϕ subset from a set `s` is:

```
(defun f (s)
  (if (ur-elementp s)
      nil
      (if (φ (scar s))
          (scons (scar s) (f (scdr s)))
          (f (scdr s)))))
```

The test on whether `s` is an `ur-elementp` is the standard way to enforce the non-set convention. This test recognizes `nil`, but also all other ur-elements, as the base case of the recursion. They are all treated equivalently.

Otherwise, `s` is a non-empty set and we define f in terms of `(scar s)` and `(scdr s)`. We have not previously mentioned these functions because they are not pure "set theory" functions: they do not admit = as a congruence relation. It is best to think of `scar` and `scdr` as working on a presentation of a set. `Scar` returns the first element presented and `scdr` returns the set containing all the others. But a set may have multiple presentations. For example, `(= '(1 2) '(2 1))` but `(scar '(1 2))` is `1` while `(scar '(2 1))` is `2`.

In fact, `ur-elementp`, `scar`, `scdr`, and `scons` are exactly analogous to `atom`, `car`, `cdr`, and `cons`, except that conses marked with `:ur-cons` are treated as atoms. Ignoring the issue raised by `:ur-cons`, the definition of f above is

```
(defun f (s)
  (if (atom s)
      nil
      (if (φ (car s))
          (cons (car s) (f (cdr s)))
          (f (cdr s)))))
```

It thus computes a list, not a set.[7] It may seem counterintuitive to prefer recursive definitions of set theory functions in terms of functions that expose the underlying representation. But this is a deliberate choice and is perhaps the key discovery made in the project. (We discuss what we call "recursion by `choose`" in Appendix A. This recursive scheme is entirely set theoretic in nature but makes inductive proofs a little more awkward.)

[7] A sentence to which a mathematically sophisticated teenager might respond with "Duh." Sets are lists in ACL2. Of course it computes a list. It computes the list in an order determined by the presentation of its arguments.

The main appeal of using scar and scdr is that it makes it possible to prove inductively the fundamental theorems about newly defined recursive set theory functions by proving the analogous theorem about list processing. The latter kind of theorems are ACL2's "bread and butter."

16 The Defmap Macro

We have defined a macro to make it easy to define functions corresponding to two common set builder forms. Each use of the macro not only defines a function but also proves certain theorems about the new function.

(defmap f (v_1 ... v_k) **:for** x **:in** v_i **:such-that** ϕ): defines (f v_1 ... v_k) to be $\{x \mid x \in v_i \wedge \phi\}$.

(defmap f (v_1 ... v_k) **:for** x **:in** v_i **:map** ϕ): defines (f v_1 ... v_k) to be $\{e \mid \exists x(x \in v_i \wedge e = \phi)\}$.

In both **defmap** forms, the variable symbol x should not be among the $v_1, \ldots, v_k$ and is expected to be used in ϕ. Conversely, v_i must be among the $v_1, \ldots, v_k$ and should not be used in ϕ, since the definition of f recurs through it. The user is expected to provide the function name, f, and to provide the formal variables $v_1, \ldots, v_k$. The formals other than v_i are presumably used in ϕ.

Defmap is a macro that expands into a sequence of ACL2 forms consisting of a definition and some theorems. These theorems are commonly needed consequences of the set builder notation but are by no means sufficient for all uses of the newly defined functions.

In the following generic examples, suppose **phi** is a function of three arguments and suppose that we know that it admits = as a congruence in each argument position.

The expansion of

```
(defmap phi-subset (a s b) :for x :in s :such-that (phi a x b))
```

is shown below. The theorems should always be provable automatically, provided **phi** admits = as an equivalence relation.

```
(encapsulate
 nil

; Define the function to extract the elements, x, of s such that
; (phi a x b).

 (defun phi-subset (a s b)
   (if (ur-elementp s)
       nil
     (let ((x (scar s)))
       (if (phi a x b)
           (scons (scar s)
                  (phi-subset a (scdr s) b))
         (phi-subset a (scdr s) b)))))

; It returns a set.

 (defthm setp-phi-subset
   (setp (phi-subset a s b)))

; It is an ur-element precisely when it is nil.
```

```
(defthm ur-elementp-phi-subset
  (equal (ur-elementp (phi-subset a s b))
         (equal (phi-subset a s b) nil)))

; X is a member of it only if x is in s and phi holds.

(defthm mem-phi-subset
  (equal (mem x (phi-subset a s b))
         (and (phi a x b) (mem x s)))
  :otf-flg t)

; It is a subset of s.

(defthm subsetp-phi-subset
  (subsetp (phi-subset a s b) s))

; It admits = as a congruence in all three arguments.

(defx :strategy :congruence (phi-subset a s b) 1)
(defx :strategy :congruence (phi-subset a s b) 2 :method :subsetp)
(defx :strategy :congruence (phi-subset a s b) 3)

; Elements of its subsets satisfy phi also.  This lemma is useful in
; mitigating the system's limited heuristics for dealing with free-
; variables in the hypotheses of lemmas, such as s in mem-phi-subset.

(defthm mem-phi-subset-corollary
  (implies (and (subsetp s1 (phi-subset a s b))
                (mem x s1))
           (phi a x b)))

; Its cardinality is no greater than that of s.

(defthm cardinality-phi-subset
  (<= (cardinality (phi-subset a s b))
      (cardinality s))
  :rule-classes :linear)

; Union distributes over it.

(defthm union-phi-subset
  (= (phi-subset a (union s1 s) b)
     (union (phi-subset a s1 b)
            (phi-subset a s b))))

; Intersection distributes over it.

(defthm intersection-phi-subset
  (= (phi-subset a (intersection s1 s) b)
     (intersection (phi-subset a s1 b)
                   (phi-subset a s b)))))
```

The expansion of

```
(defmap phi-image  (a s b) :for x :in s :map      (phi a x b))
```

is shown below. The reader will note that some of the theorems below are weaker than the corresponding theorems above.

```
(encapsulate
 nil

 (defun phi-image (a s b)
   (if (ur-elementp s)
       nil
     (let ((x (scar s)))
       (scons (phi a x b)
              (phi-image a (scdr s) b)))))

 (defthm setp-phi-image (setp (phi-image a s b)))

 (defthm ur-elementp-phi-image
   (equal (ur-elementp (phi-image a s b))
          (ur-elementp s)))

 (defthm weak-mem-phi-image
   (implies (and (mem x s) (= x1 (phi a x b)))
            (mem x1 (phi-image a s b))))

 (defthm subsetp-phi-image
   (implies (subsetp s1 s)
            (subsetp (phi-image a s1 b)
                     (phi-image a s b))))

 (defx :strategy :congruence (phi-image a s b) 1)
 (defx :strategy :congruence (phi-image a s b) 2 :method :subsetp)
 (defx :strategy :congruence (phi-image a s b) 3)

 (defthm cardinality-phi-image
   (<= (cardinality (phi-image a s b))
       (cardinality s))
   :rule-classes :linear)

 (defthm union-phi-image
   (= (phi-image a (union s1 s) b)
      (union (phi-image a s1 b)
             (phi-image a s b))))

 (defthm intersection-phi-image
   (subsetp (phi-image a (intersection s1 s) b)
            (intersection (phi-image a s1 b)
                          (phi-image a s b)))))
```

In the current set theory book, these theorems are proved inductively when the **defmap** form is introduced. In future work the theorems should be derived “for free” by functional instantiation from theorems proved about one the generic “such-that” and one generic “map” functions. That will be easier to extend and will make the proofs faster and more reliable.

A Recursion by `Choose`

The definition of set `union` provided in ACL2 set theory is

```
(defun union (a b)
  (if (ur-elementp a)
      (sfix b)
      (scons (scar a) (union (scdr a) b))))
```

It is defined in terms of the functions `scar` and `scdr` which do not admit `=` as a congruence. As noted, we prefer such recursive definitions because `scar`/`scdr` recursions "ripple" through of nests of similarly defined functions during induction. Induction and recursion are extremely important in this setting since they embody the basic set builder notation and allow us to deduce properties of the constructed sets.

In this appendix we illustrate an alternative approach to recursive definition which is "more pure" from the set theoretic perspective.

Here is a definition of set union defined in a purely set theoretic way. By that I mean that the definition in no way exposes the underlying list representation of sets. The key is to use `(choose s)` as a way to select an element from a set and to delete it from the set upon recursion. The function `abc` (which stands for "All But Choose") deletes the element from the set, using `diff`.

```
(defun abc (s)
  (diff s (brace (choose s))))

(defun un (a b)
  (declare (xargs :measure (cardinality a)))
  (if (ur-elementp a)
      (sfix b)
    (scons (choose a) (un (abc a) b))))
```

The definition is admitted because the cardinality of the set `a` decreases.

Now consider proving

```
(defthm un-assoc
  (= (un (un a b) c) (un a (un b c))))
```

The corresponding theorem about `union` is proved by induction in ACL2 set theory. The proof is trivial: as easy as the inductive proof of the associativity of `append` — indeed, it is the associativity of `append`.

But an inductive proof of `un-assoc` is more complicated until certain groundwork is established. The inductive hypothesis of an induction on `a` is `(= (un (un (abc a) b) c) (un (abc a) (un b c)))`. The trick is getting `(un (un a b) c)` to simplify to something involving `(un (un (abc a) b) c)`. The problem is that the inner `un` expression, `(un a b)`, recurs by removing `(choose a)` from its first argument. But the outer `un` expression recurs by removing `(choose (un a b))`, which is generally different from `(choose a)`. This makes it very difficult to deal with `un` inductively because as similarly defined nested applications expand, they decompose successive outputs in different ways, i.e., recursions do not "ripple up" in the `choose` style because different elements are chosen at each level.

The key to proving theorems about `un` is to prove the following lemma first:

```
(= (un (scons e a) b)
   (scons e (un a b))).
```

This basically eliminates the use of the recursive definition and its "preference" to use `choose`.

The key to proving this is to prove the following fundamental fact:

```
(implies (not (ur-elementp a))
         (= (scons (choose a) (abc a)) a))
```

That is, any set can be thought of as the `scons` of its `choose` and its `abc`.

In order prove the key theorem relating `un` and `scons` it is necessary to consider both `(choose (scons e a))` and `(abc (scons e a))`. The weak `choose-scons` property is sufficient to handle the first. How the second simplifies depends on which element is chosen by the first. The following lemma describes both cases, without introducing mention of our total order.

```
(defthm choose-dichotomy
  (implies (not (ur-elementp a))
           (or (and (= (choose (scons x a)) x)
                    (= (abc (scons x a))
                       (if (= x (choose a))
                           (abc a)
                         a)))
               (and (= (choose (scons x a)) (choose a))
                    (= (abc (scons x a))
                       (if (= x (choose a))
                           (abc a)
                         (scons x (abc a))))))))
```

If this lemma were part of our set-theory book (it is not, but I have proved it in a related book exploring recursion by `choose`), and coded so as to be an effective rewrite rule, then proofs about functions defined using recursion by `choose` are no more difficult than those defined using `scar` and `scdr`.

In particular, I have a book that configures ACL2 so that it is easy to prove:

```
(defthm associativity-of-un
  (= (un (un a b) c) (un a (un b c))))

(defthm commutativity-of-un
  (= (un a b) (un b a)))

(defthm union-un
  (= (un a b) (union a b)))
```

suggesting that we can deal reasonably well with the alternative scheme.

I personally prefer the `scar` and `scdr` recursion and induction, because it so completely converts the set theory problems into the domain where ACL2 is most powerful. The = relation is responsible for ignoring duplications and order, and so we might as well regard our sets as ordered when it makes inductive proofs easier.

B Background of the Project

This work was supported in part by Compaq Systems Research Center, Palo Alto, Ca. The goal of the project was to investigate the provision of mechanical proof support for Leslie Lamport's TLA, leveraging off the ACL2 theorem prover. Therefore, we souht to translate some TLA formulas into ACL2 formulas such that if the latter are theorems of ACL2, the former are theorems of TLA.

TLA is based on set theory. Indeed, every object in its domain is a set: numbers, sequences, functions, etc., are all represented as sets. However, TLA does not spell out which sets represent which objects. Infinite sets exist. Logically speaking, the TLA syntax is untyped and all functions and operators are total. Thus, some syntactically well-formed questions, such as $3 \in$ `"ABC"` and $|\{3,$ `"ABC"`$\}|$, have unspecified answers. The TLA community deals with this by never expecting answers to such questions.

ACL2 is based on inductively constructed objects. The construction of rational and complex numbers, characters, strings, symbols, and ordered pairs are described in complete detail so that the value of any ground term can be computed. Indeed, ACL2 is a subset of Common Lisp. Every object is finite. The syntax is untyped and all functions are total.

Aside from small fragments of the two logics, e.g., propositional calculus and arithmetic, it is impossible to imagine a translation from TLA to ACL2 without first providing some notion of "set" in ACL2.

But how do we reconcile the different ontological perspectives of TLA and ACL2? Consider the integers. Are they sets? TLA says "yes." We could formalize an ACL2 set theory providing embedded arithmetic. That is, certain sets would be known to represent numbers and set theoretic functions would be defined as the analogues of the arithmetic functions. We could optionally choose to expose the embedding or not. I discuss this approach further later. This approach would put the TLA ACL2 user at a disadvantage because ACL2 either deals with arithmetic "at arm's length" or else spends a lot of reasoning power converting the problem into ACL2 native arithmetic.

I felt it was advantageous to construct a set theory that was natural to ACL2. Here, sets may contain such ACL2 objects as numbers and symbols. The set theory user can thus take advantage of many books dealing with useful theories such as bit vectors and floating point. How this set theory is related to the TLA project is a separate issue. Perhaps it will merely serve as a witness for the constrained introduction of "under-specified" TLA set theory functions. Perhaps the translator from TLA to ACL2 would deal with most of these issues. We discuss this below. But it seems clear that (a) we need some natural notion of set theory in ACL2 to even consider translation from TLA and (b) this notion ought to be of general use to the ACL2 community simply so it does not become an unsupported backwater. By building such a set theory I hope that the wider ACL2 community will find it useful and elaborate and develop it.

Returning to the TLA perspective, I imagined two levels of "users" for this set theory. A *TLA expert*, probably working with me, could use this set theory and other aspects of ACL2 to produce a translator from TLA to ACL2 proof obligations. This would involve proving additional theorems, most probably at the set theory level, having to do with the implementation of additional TLA abstractions and the codification of TLA well-formedness conditions to insure that the ACL2 set theory is not used to answer questions that the TLA user "should" never ask. Finally, the *end user* (or *TLA user*) would attempt to prove theorems produced by the translator. This will involve both the fundamental theorems proved by me and the well-formedness conditions codified by the TLA expert. My goal was to produce a set theory that the TLA expert could use and that would yeild manageable proof obligations for the end user.

The nature of ACL2 is such that *one must expect the end user to have some familiarity with ACL2* in order to "drive" the theorem prover. That is, the envisioned system would not be "push button." It will sometimes be necessary for the end user to guide ACL2 with hints of various sorts, e.g., "induct on `z`," "use this instance of `lemma-23`," or "expand the definition of `cardinality`." The challenge was to free the users from dealing with representational issues.

Elementary set theory operations on finite sets are straightforward to translate into ACL2 set theory.

But there are a variety of problems as soon as we step away from the algebraic fragment of set theory. I believe it is better to handle these problems in the translation than it is to try to use ACL2 so far outside its intended domain. I comment here on some of the anticipated problems of translating TLA set theory into ACL2.

The finiteness of our sets raises a problem. While it is probably easy to limit most TLA constructed sets to finite ones, there are several built-in infinite sets, such as the set of all natural numbers, **Nats**. **Nats** is simply not a set in our set theory. The translation of something like $A \cap$ **Nats** is `(nat-subset A)`, where `(defmap nat-subset (s) :for x :in s :such-that (natp x))`.

The absence of quantification is a problem. If the quantification is over a finite domain, it will be possible to use `defmap` functions. The form $\forall x(x \in s \rightarrow \phi(x))$ is `(= s (phi-subset s))`, where `phi-subset` is the obvious `defmap` function. Similarly, $\exists x(x \in s \wedge \phi(x))$ is `(not (= nil (phi-subset s)))`.

Quantification over infinite domains can sometimes be handled by exploiting the deduction law. For example, $(\forall x \phi) \rightarrow \theta$ is inter-provable with $\phi \vdash \theta$, i.e., by adding ϕ as an axiom and proving θ. ACL2 also provides `defun-sk`, by which full first order quantification can be handled, albeit manually. Generally speaking, such tricks as these produce unsatisfactory results because ACL2 is really not designed to handle their frequent use.

A potential problem arises from the facts that ACL2 set theory provides ur-elements and TLA does not. In ACL2 one can prove that `2` is not a member of `'(1 (0))`, while in TLA, 2 may be a member of $\{1, \{0\}\}$ because it is not specified which set 2 is, so it is possible that 2 is $\{0\}$. (We know 2 is not 1.)

Similarly, in ACL2 one can prove that `(+ 3 '(0 1))` is `3` because `+` "expects" numeric arguments, non-numeric ones are treated as `0`, and the set `'(0 1)` is definitely not a number. In TLA, the set $\{0, 1\}$ might be a number.

ACL2 guards are designed to handle this problem. Guards allow the user to specify pre-conditions on functions. These pre-conditions do not affect the logical semantics of the function definition. But they interact with a "guard verification" mechanism whereby one can prove that the pre-conditions on all the functions used in a formula are satisfied in that formula. We call such a formula a *compliant* formula.

Guards were introduced into ACL2 to allow us to avoid the partially defined functions of Common Lisp (such as `car`) while still claiming a relationship with Common Lisp. For example, the guard on the list concatenation function, `append`, is that the first argument must be a "true-list," a list ending in `nil`. That is because in Common Lisp, `append` terminates when its first argument is `nil` and otherwise `cdr`s it. The `cdr` of an atom (other than `nil`) is undefined in Common Lisp. But in ACL2 the `cdr` of all atoms is `nil`. In ACL2 we can prove

```
(equal (append (append a b) c)
       (append a (append b c))).
```

That is, `append` is associative, for all inputs. An instance of this theorem is

```
(equal (append (append 1 2) 3)
       (append 1 (append 2 3))),
```

which, of course, is also an ACL2 theorem. This formula is not "true" in Common Lisp. It will cause most Common Lisps to "break" or signal an error.

While the associativity of `append` is a theorem of ACL2, it is not compliant. The guards for `append` are not satisfied in this formula. However, consider the following formula:

```
(implies (and (true-listp a)
              (true-listp b))
         (equal (append (append a b) c)
                (append a (append b c)))).
```

This formula is compliant. The proof that it is compliant is not immediate: one must show that if `a` and `b` are are true-lists then so is `(append a b)`, since `(append a b)` is used as the first argument of an `append`. The process of guard verification proves the necessary theorems to establish that every guard encountered in the formula is satisfied in the context in which it is encountered. So the formula is compliant.

It is also a theorem. Indeed, it is trivial to prove because it follows from the much stronger theorem stating the unconditional associativity of `append`.

We have shown that if a theorem is compliant then the behavior of its guarded functions outside their guarded domains is irrelevant. That is, the formula would be a theorem even in a weaker logic in which functions were partial (see "An Industrial Strength Theorem Prover for a Logic Based on Common Lisp,

M. Kaufmann and J S. Moore, *IEEE Transactions on Software Engineering* **23**(4), April 1997, pp. 203–213). Thus, by providing the appropriate guards for functions, it is possible to identify a subset of the ACL2 theorems that are known to be theorems in a weaker logic.

The translator I envision will generate formulas containing symbols in a new package, `"TLA"`. ACL2's set theory functions will be shadowed out of this package. TLA functions will be defined to be equal to their ACL2 counterparts but guards will be imposed on the TLA functions.

I imagine defining the symbol `+` in the `TLA` package as follows:

```
(defun tla::+ (i j)
  (declare (xargs :guard (and (rationalp i) (rationalp j))))
  (acl2::+ i j))
```

Logically speaking, `(tla::+ i j)` is `(acl2::+ i j)`. (If you make the TLA package the "current package" then the formula `(+ i j)` will be read as `(tla::+ i j)` and `(tla::+ i j)` is printed as `(+ i j)`.) So proving theorems about the one is as easy as proving theorems about the other and we can arrange the current package to make the output favor whichever set of symbols we see most often.

But a formula involving `(tla::+ i j)` is compliant only if it can be proved that the two arguments `i` and `j` are rational numbers. As long as TLA and ACL2 agree on what addition means on the rationals and we restrict our attention to formulas that are compliant with the TLA guards, then it does not matter whether TLA and ACL2 differ on what addition means on non-rationals.

Similarly, a guarded version of `tla::mem` could be provided, in which the guards are sufficient to imply that ACL2's `mem` computes answers that are consistent with TLA's. Thus, it would be possible to use ACL2's "over-specified" set theory to prove theorems about TLA's "under-specified" theory. Most interestingly, guards allow the proof and use of lemmas that are not TLA-compliant, as long as the final theorem is compliant.

Much of TLA's "under-specified" theory could be formalized now in ACL2 by introducing constrained functions (in the `"TLA"` package) for `setp`, `mem`, `union`, `choose`, `functionp`, `apply`, etc. These constrained symbols would be witnessed by the ACL2 set theory functions. Then selected theorems about the ACL2 functions would be exported as the axioms about the TLA symbols.

An alternative, and preferred, way of handling this problem is to do so syntactically at the TLA level: simply refuse to translate any TLA formula that violates the implicit "guards" on TLA functions.

C Dead Ends and False Starts

I briefly describe the research that was done, including some "dead ends" and "false starts." These explorations have heavily influenced the final theory and may help the reader accept some of the decisions made.

This project has consumed four weeks of my time during the 1999 calendar year: one week at SRC in April, one week at SRC in July, and two weeks in November, in Austin.

I took the view that I was trying to develop a set theory that included the high level objects of TLA, e.g., numbers, strings, functions, etc., as well as sets. The "customer" or "user" of this set theory would want to reason formally about these objects, using ACL2 theorems proved by me. As noted in the introductory remarks, I imagined at least three levels of users: the ACL2 expert who constructs the basic set theory, the TLA expert who uses it as a target for translation, and the end user who tries to prove translated theorems.

My early thinking was that it did not matter if the low-level proofs were awkward and required great ACL2 hacking skills, if the end result was a set theory in which high-level proofs of simple theorems were easy.

Lisp lists provide a functionality similar to sets. Identity is checked by the function `equal`, membership in a list by `member`, and the subset relation between two lists by `subsetp`. Both `member` and `subsetp` use `equal` to compare elements. Thus, 2 is a `member` of `(1 2 3)`. But `(1 2)` is not a `member` of `((2 1))`, because `(1 2)` and `(2 1)` are un-`equal` lists. Thus, the "set theory" native to Lisp — where one just ignores duplications and order — is "flat" in the sense that it works only for sets of atoms. When "sets" are used as elements, distinct representations of the "identical" sets are not identified. So we are forced to generalize Lisp's native "set theory."

In set theory, two sets are equal if and only if they are subsets of one another. If we *define* set equality this way, then equality, membership and subset are mutually recursive: set equality is defined in terms of subset, membership is iterated set equality, and subset is iterated membership. ACL2 supports mutual recursion. But it is mathematically complicated because of its effect on induction. To prove an inductive theorem about one function in a clique of mutually recursive functions, one must often prove a conjunction of related theorems about the other functions of the clique. While ACL2 can often manage the proofs, the user must state the conjunction of theorems in order for it to be strong enough to be provable. I found this often difficult, especially in the early going when nothing but the definitions of the functions are available.

I explored ways around this problem. My eventual solution, which was supported by some work being done concurrently by Pete Manolios, was to introduce the notion of canonical forms. Without engaging in mutual recursion it is possible to

- Define a total order on the objects used in sets.
- Canonicalize lists so that their elements are presented in this order and without duplicates.
- Define set equality to compare sets in canonical form.
- Define membership as iterated set equality.
- Define subset as iterated membership.
- Prove that two sets are set equal iff they are subsets of one another.

This program actually has at least two interpretations and I explored both. The interpretation I initially favored was to keep lists representing sets in canonical form all the time. That is, the basic set constructor, e.g., `(scons e x)` which adds `e` as an element to set `x`, inserts `e` (if it is not already there) at the position in `x` at which it belongs. This has the powerful attraction that set equality is `equal`.

I found this approach to complicate set construction to a degree out of proportion to its merits. In particular, functions like `union` and `intersection`, which are quite easy to reason about in the list world (where order and duplication matter but are simply ignored), become quite difficult to reason about in the set world, where most of the attention is paid to the sorting of the output with respect to the total ordering. In the end I abandoned this approach and adopted a second interpretation of the program above: lists representing sets are created and manipulated in non-canonical form and are canonicalized only for determining whether two sets are equal. This was quite effective because

- the set equality relation can be proved to be an equivalence,
- set equality can be proved to be a congruence relation maintained by the "natural" definitions of Lisp `union`, etc., and
- the "natural" definitions of `union`, etc., while possibly producing non-canonical sets, are extremely simple and easy to reason about via induction.

Another question that I investigated was: what are the "ur-elements" of my set theory? My first attack was to formalize hereditarily finite sets: finite sets built entirely from the empty set. My early thinking was that the details of the set theory were irrelevant to the user, since the high level objects with which the user would deal — numbers, sequences, functions, etc., — would be abstractly represented. According to this thinking, proofs about these high level objects would be conducted more or less algebraically, using theorems proved by me.

However, I found the use of hereditarily finite sets to be too cumbersome.

- Concrete examples of sets representing naturals, pairs, etc., were practically impossible for me to read and comprehend. Here is 3 in the von Neumann representation of the naturals $\{\{\{\{\}\}\{\}\}\{\{\}\}\{\}\}$. It was hard to test definitions and conjectures.

- The need to embed everything as sets forced ACL2 to spend its resources unraveling the embedding rather than dealing with the gist of the user's problem.

The last point is the more important. Consider a simple user-level theorem such as that $|a| < |a| + |b|$, if $b \neq \{\}$. This might be translated to ACL2 as

```
(implies (not (set-equal b (empty-set)))
         (set-< (set-card a)
                (set-+ (set-card a) (set-card b))))
```

Obviously, `set-card` is the function that takes a set and returns its cardinality. But its cardinality is not an ACL2 natural number but a set!. Therefore, we have to introduce set analogues of the arithmetic functions, e.g., `set-+` and `set-<` which operate on sets encoding natural numbers.

To prove this theorem, ACL2 must use foundational lemmas allowing it to rewrite the formula to:

```
(implies (not (equal b nil))
         (< (card a) (+ (card a) (card b))))
```

The foundational lemmas required include

```
(implies (and (set-natp a)
              (set-natp b))
         (iff (set-< a b)
              (< (nat a) (nat b))))

(implies (and (set-natp a)
              (set-natp b))
         (equal (nat (set-+ a b)
        (+ (nat a) (nat b)))))

(equal (nat (set-card a))
       (card a))
```

The first says that if `a` and `b` are sets representing natural numbers, then `(set-< a b)` is equivalent to `(< (nat a) (nat b))`, where `nat` is the mapping from sets representing naturals to naturals. This lemma allows us to "trade in" `set-<` for ACL2's native `<` operator. The second lemma says that the natural corresponding to `(set-+ a b)` is the sum of the naturals corresponding to `a` and `b`. The third says that the set returned by `(set-card a)` represents the natural returned by `card`. `Card` is the ACL2 function that computes the cardinality of a set, returning an ACL2 natural rather than a set representing a natural.

Lemmas such as the first convert set theoretic predicates into ACL2 predicates and introduce `nat` or, in general, the inverse mappings of our embedding. Lemmas such as the second and third drive these inverses down through set theory operations, turning them into native ACL2 functions. But such lemmas generally have hypotheses; to apply them, ACL2 must back chain. To apply the first, for example, ACL2 must prove that both `(set-card a)` and `(set-+ (set-card a) (set-card b))` return sets representing naturals.

Notice that a lot of reasoning must happen merely to convert the set theory problem to ACL2's "natural" domain. I decided this was counterproductive. The translation from TLA to ACL2 set theory should deal with these problems. In the final view of this project, I saw my job as to produce a set theory that was "natural" to ACL2's mode of reasoning, so that its power would be spent at the core of the user's problem, not on getting down there.

A final issue I have investigated is the manner in which recursive functions on sets are defined: shall we use `scar`/`scdr` recursion or `choose`/`abc` recursion. I have already commented at length about this and so will not belabor it; but it was an important aspect of the research done here.

D Some Sample Theorems

Here are some of the theorems available in `set-theory.lisp`.

Theorem 1.
```
(subsetp x x)
```

Theorem 2.
```
(implies (and (subsetp a b)
              (subsetp b c))
         (subsetp a c))
```

Theorem 3.
```
(implies (and (mem e a)
              (subsetp a b))
         (mem e b))
```

Theorem 4.
```
(implies (and (setp a)
              (setp b))
         (iff (= a b)
              (and (subsetp a b)
                   (subsetp b a))))
```

Theorem 5.
```
(not (mem e e))
```

Theorem 6.
```
(iff (mem e (union a b))
     (or (mem e a) (mem e b)))
```

Theorem 7.
```
(subsetp a (union a b))
```

Theorem 8.
```
(subsetp a (union b a))
```

Theorem 9.
```
(implies (subsetp a1 a2)
         (subsetp (union a1 b)
                  (union a2 b)))
```

Theorem 10.
```
(= (union a b) (union b a))
```

Theorem 11.
```
(= (union (union a b) c)
   (union a (union b c)))
```

Theorem 12.
```
(equal (mem e (intersection a b))
       (and (mem e a) (mem e b)))
```

Theorem 13.
```
(subsetp (intersection a b) a)
```

Theorem 14.
```
(implies (subsetp a1 a2)
         (subsetp (intersection a1 b)
                  (intersection a2 b)))
```

Theorem 15.
```
(= (intersection a b) (intersection b a))
```

Theorem 16.
```
(= (intersection (intersection a b) c)
   (intersection a (intersection b c)))
```

Theorem 17.
```
(<= (cardinality a)
    (cardinality (union a b)))
```

Theorem 18.
```
(<= (cardinality (intersection a b))
    (cardinality a))
```

Theorem 19.
```
(iff (= (cardinality a) 0)
     (ur-elementp a))
```

Theorem 20.
```
(implies (not (intersection a b))
         (= (cardinality (union a b))
            (+ (cardinality a)
               (cardinality b))))
```

Theorem 21.
```
(= (mem (choose a) a)
   (not (ur-elementp a)))
```

Theorem 22.
```
(or (= (choose (scons e a)) e)
    (= (choose (scons e a)) (choose a)))
```

Theorem 23.
```
(= (choose (scons e nil)) e)
```

Theorem 24.
```
(implies (and (= (cardinality x) 1)
              (mem e x))
         (= (scons e nil) x))
```

Theorem 25.
```
(iff (mem e (diff a b))
     (and (mem e a)
          (not (mem e b))))
```

Theorem 26.
```
(implies (subsetp a1 a2)
         (subsetp (diff a1 b)
                  (diff a2 b)))
```

Theorem 27.
```
(implies (and (subsetp a b)
              (subsetp b c))
         (= (union (diff b a)
                   (diff c b))
            (diff c a)))
```

Theorem 28.
```
(<= (cardinality (diff a b))
    (cardinality a))
```

Theorem 29.
```
(= (intersection (diff b c) a)
   (diff (intersection a b) c))
```

Theorem 30.
```
(= (hd (pair x y)) x)
```

Theorem 31.
```
(= (tl (pair x y)) y)
```

Theorem 32.
```
(pairp (pair x y))
```

Theorem 33.
```
(implies (pairp a)
         (= (pair (hd a) (tl a)) a))
```

Theorem 34.
```
(iff (= (pair x1 y1)
        (pair x2 y2))
     (and (= x1 x2)
          (= y1 y2)))
```

Theorem 35.
```
(implies (functionp f)
         (functionp (except f x v)))
```

Theorem 36.
```
(= (apply (except f x v) y)
   (if (= x y)
       v
     (apply f y)))
```

Theorem 37.
```
(implies (pairsp f)
         (= (domain (except f x v))
            (scons x (domain f))))
```

Theorem 38.
```
(subsetp (range (except f x v))
         (scons v (range f)))
```

Theorem 39.
```
(= (domain (union f g))
   (union (domain f) (domain g)))
```

Theorem 40.
```
(= (range (union f g))
   (union (range f) (range g)))
```

Theorem 41.
```
(<= (cardinality (domain f))
    (cardinality f))
```

Theorem 42.
```
(<= (cardinality (range f))
    (cardinality f))
```

Theorem 43.
```
(implies (and (functionp a)
              (functionp b)
              (not (intersection (domain a)
                                 (domain b))))
         (functionp (union a b)))
```

Theorem 44.
```
(implies (not (intersection (domain a)
                            (domain b)))
         (not (intersection a b)))
```

Theorem 45.
```
(implies (functionp f)
         (iff (mem e f)
              (and (pairp e)
                   (mem (hd e) (domain f))
                   (= (tl e)
                      (apply f (hd e))))))
```

Theorem 46.
```
(= (domain (restrict f s))
   (intersection s (domain f)))
```

Theorem 47.
```
(implies (and (mem x (domain f))
              (mem x s))
         (= (apply (restrict f s) x)
            (apply f x)))
```

Theorem 48.
```
(implies (integerp delta)
         (= (cardinality (shift i j f delta))
            (if (and (integerp i)
                     (integerp j)
                     (<= i j))
                (+ 1 (- j i))
              0)))
```

Theorem 49.

```
(implies (and (functionp a)
              (functionp b)
              (not (intersection (domain a)
                                 (domain b))))
         (= (apply (union a b) x)
            (if (mem x (domain a))
                (apply a x)
              (apply b x))))
```

Theorem 50.

```
(= (apply (shift i j f delta) x)
   (if (mem x
            (domain (shift i j f delta)))
       (apply f (- x delta))
     nil))
```

Theorem 51.

```
(implies (and (sequencep r)
              (sequencep s))
         (= (apply (concat r s) i)
            (let ((n (cardinality r)))
       (cond
              ((and (integerp i)
                    (< 0 i))
               (if (<= i n)
                   (apply r i)
                 (apply s (- i n))))
              (t nil)))))
```

Theorem 52.

```
(implies (and (sequencep a)
              (sequencep b)
              (sequencep c))
         (= (concat (concat a b) c)
            (concat a (concat b c))))
```